CEYX-Version 15.I : une initiation

J.M. Hullot

INRIA

CENTRE DE ROCQUENCOURT

Rapports Techniques

N° 44

CEYX - Version 15

I : UNE INITIATION

Jean-Marie HULLOT

Février 1985

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél. : (1) 39 63 55 11

INRIA
Domaine de Voluceau
Rocquencourt
78153 Le Chesnay Cedex

CEYX - Version 15

I: Une Initiation

Jean-Marie Hullot

Eté 1984

Résumé: *CEYX est un ensemble d'outils permettant de faciliter la tâche des programmeurs LISP pour définir et manipuler des structures arbitraires. Aux structures définies en CEYX sont associées des espaces sémantiques, organisés hiérarchiquement, dans lesquels sont rangées les fonctions de manipulation propres à ces structures. Un style de programmation orientée objets à la SmallTalk est ainsi rendu possible.*

Nous présentons dans ce document, les constructions élémentaires du système.

Abstract: *CEYX is a set of tools allowing to define and manipulate arbitrary data structures using the LISP programming language. Semantical properties are associated to CEYX structures: they are the basic actions that can be performed on such structures. Moreover, structures are arranged by families in a hierarchical manner.*

In this report, we present the basic constructions of the system.

CEYX était fils de LUCIFER,
le conducteur des Astres et de la Lumière,
l'étoile qui fait naître le jour,
et la joie brillante de son père illuminait son visage.

Ovide

Avertissement

Ce document constitue une introduction à CEYX. Une connaissance minimum de LISP est requise. CEYX est écrit en LE_LISP, qui possède un excellent manuel disponible auprès de l'INRIA:

- *LE_LISP de l'INRIA, Le Manuel de Référence* (J. Chailloux).

On trouvera une description complète de CEYX dans un autre rapport INRIA:

- *Programmer en CEYX* (J.-M. Hullot),

et la documentation de processeurs spécialisés dans:

- *VPRINT, Le Composeur CEYX* (G. Berry, J.-M. Hullot),
- *CXYACC et LEX-KIT* (G. Berry, B. Serlet).

CHAPITRE 1

Les Records LISP

Nous introduisons dans ce chapitre des constructions permettant de définir de nouvelles structures de données en LISP. Il s'agit essentiellement de constructions simples qui devraient être comprises sans difficultés par un lecteur un peu habitué à LISP. Ce chapitre peut être considéré comme une initiation à CEYX dans la mesure où il introduit à un style de programmation et à un ensemble de notations qui sont de mise en CEYX.

1.1 Introduction

Nous proposons une construction permettant de manipuler en LISP des structures de données à éléments nommés analogues aux records de Pascal. Cherchons par exemple à modéliser la notion de point matériel en mouvement dans un plan cartésien. Il s'agit d'objets possédant les deux champs x et y, qui sont les coordonnées du point, et les deux champs vx et vy qui sont les vitesses par rapport aux axes de coordonnées. Pour manipuler de tels objets en LISP, on commencera par définir une fonction Point créant un objet LISP composite mémorisant ces valeurs. Une solution minimale en utilisation mémoire et en rapidité moyenne d'accès aux champs est:

```
? (de Point (x y vx vy)
      (cons (cons x y) (cons vx vy)))
= Point
? (setq point1 (Point 0 0 0 0))
= ((0 . 0) 0 . 0)
```

Il nous est alors possible d'inspecter et de modifier les composantes x,y, vx et vy d'une instance de Point par les chaines de car, cdr, rplaca, rplacd idoines. Une bonne solution consiste à définir des fonctions d'accès aux champs en lecture et en écriture et de ne plus accéder aux instances de Points que par ces fonctions.

```
? (de get-Point-x (point) (caar point))
= get-Point-x
? (de put-Point-x (point val) (rplaca (car point) val)))
= put-Point-x
? (put-Point-x point1 1)
= (1 . 0)
? point1
= ((1 . 0) . (0 . 0))
? (get-Point-x point1)
= 1
```

et de la même façon des fonctions pour les champs y, vx et vy. Le problème de la manipulation des Points semble ainsi bien résolu au prix de l'écriture d'un certain nombre de fonctions. Ceci nous permet de faire nos premiers pas dans l'écriture d'un programme sur les Points. Malheureusement, nous nous apercevons au bout d'un certain temps que nous voulons en fait manipuler des points nommés et qu'ainsi un nouveau champ appelé nom devient nécessaire. Il nous faut alors redéfinir la fonction Point prenant cette fois-ci cinq arguments et toutes les fonctions d'accès puisque la structure de représentation des Points en tant qu'arbre de cons a été modifiée. On notera que tous les appels à la fonction Point dans le code devront être également modifiés puisque cette fonction prend maintenant un argument supplémentaire. Le but de la construction que nous proposons est de faciliter ce genre de manipulations, en

introduisant de nouvelles constructions dans LISP.

Pour définir la structure Point sous CEYX, il suffit d'invoquer:

```
? (defrecord Point x y vx vy)
= Point
```

Une instance de Point sera alors créée par:

```
? (setq point1 (omakeq Point x 0 y 0 vx 0 vy 0))
= ((0 . 0) 0 . 0)
```

De plus à l'invocation de la forme defrecord, les fonctions d'accès {Point}:x, {Point}:y, {Point}:vx et {Point}:vy sont automatiquement créées qui permettent d'accéder aux points en lecture et en écriture suivant qu'on leur donne un ou deux arguments.

```
? ({Point}:x point1)
= 0
? ({Point}:y point1 1)
= (0 . 1)
? ({Point}:y point1)
= 1
```

Cette fois-ci, l'effort pour rajouter un champ nom dans la structure sera minime puiqu'il suffira de remplacer l'appel de defrecord précédent par:

```
(defrecord Point nom x y vx vy)
```

Aucune autre modification ne sera nécessaire dans le code de l'utilisateur.

1.2 Définition d'un Record LISP

Un Record LISP est défini à l'aide de la forme defrecord:

(defrecord <name> <field>1 ... <field>n)

<name> est un symbole qui devient le nom de la structure ainsi définie et les <field>i sont de la forme <symbol>i ou (<symbol>i <init>i), où <symbol>i est le nom du champ et <init>i, qui est évaluée est la valeur d'initialisation du champ (voir fonction omakeq). Les champs pour lesquels la valeur d'initialisation n'est pas donnée sont initialisés à nil.

Cette construction associe au symbole <name> la définition de Lrecord à n champs de noms <symbol>i. De plus n fonctions de nom {<name>}:<symbol>i sont engendrées pour permettre d'accéder aux instances de cette structure en lecture et en écriture.

```
? (defrecord Point nom (x 0) (y 0) (vx 0) (vy 0))
= Point
```

1.3 Fonctions de Manipulation des Instances de Record

1.3.1 Création d'Instances

(omakeq <name> <fieldname>1 <val>1 ... <fieldname>n <val>n)

<name> et les <fieldname>i ne sont pas évalués, les <val>i le sont. <name> doit avoir une définition de record, c'est à dire avoir été défini par (defrecord <name> ...). Les <fieldname>i sont des noms de champ du record de nom <recordname>. Cette fonction renvoie en valeur une instance de ce record dont le champ de nom <fieldname> a pour valeur sa valeur initiale si <fieldname> n'appartient pas aux <fieldname>i, <val>i autrement.

```
? (setq point1 (omakeq Point))
= ((nil . 0) 0 0 . 0)
? ({Point}:nom point1)
= nil
? ({Point}:nom point1 'Le_Point)
= Le_Point
? ({Point}:nom point1)
= Le_Point
? (setq point2 (omakeq Point nom 'Autre_Point x 10 y 10))))
= ((Autre_Point . 10) 10 0 . 0)
? ({Point}:nom point2)
= Autre_Point
? ({Point}:x point2)
= 10
? ({Point}:vx point2)
= 0
```

1.3.2 Discrimination

(omatchq <name> <object>)

<name> n'est pas évalué, <object> l'est. <name> doit posséder une définition de Record. Cette fonction ramène une valeur différente de nil si et seulement si <object> a pu être créé par (omakeq <name> ...).

```
? (omatchq Point point1)
= t
? (omatchq Point 1)
= nil
```

1.3.3 Accès aux Champs

A côté des fonctions d'accès en lecture et en écriture engendrées automatiquement lors de la définition d'un record, nous introduisons des constructions générales d'accès aux champs. Ces constructions seront surtout utilisées lors de la définition de macros.

(ogetq <name> <fieldname> <object>

L'argument <name>, qui n'est pas évalué, doit être un symbole possédant une définition de record. Si <fieldname>, qui n'est pas évalué, n'est pas un nom de champ pour le record de nom <name>, une erreur est déclenchée. Sinon <object> doit être une instance de <name> et cette fonction ramène en valeur la valeur de son champ <fieldname>.

(oputq <name> <fieldname> <object> <object>1)

L'argument <name>, qui n'est pas évalué, doit être un symbole possédant une définition de record. Si <fieldname>, qui n'est pas évalué, n'est pas un nom de champ pour le record de nom <name>, une erreur est déclenchée. Sinon <object> doit être une instance de <name> et cette fonction remplace physiquement la valeur du champ <fieldname> par <object>1.

```
? (ogetq Point nom point1)
= Le_Point
? (oputq Point x point1 20)
= (Le_Point . 20)
? (ogetq Point x point1)
= 20
```

1.4 Les Records Extensibles ou Classes

Définissons maintenant une structure d'individu:

```
(defrecord Individu nom prenom adresse telephone)
= Individu
```

et supposons que nous voulions créeer deux nouvelles structures possédant les quatre champs du record Individu plus un nouveau champ "pointure" pour la première et un nouveau champ "condamnations" pour la seconde. Une première solution consiste à définir deux nouveaux records à cinq champs. Pour simplifier ces opérations CEYX introduit la notion de record extensible ou classe. Pour notre exemple, il suffira d'écrire:

```
? (defclass Individu nom prenom adresse telephone)
= #:Class:Individu
? (defclass {Individu}:Client pointure)
= #:Class:Individu:Client
? (defclass {Individu}:Suspect condamnations)
= #:Class:Individu:Suspect
```

Une classe LISP ou Classe est définie à l'aide de la forme:

(defclass <name>/{<superclass>}:name <field>1 ... <field>n)

Dans le premier cas, cette forme est équivalente au defrecord, sinon qu'on définit cette fois-ci un record extensible. Dans le second cas <superclass> doit avoir une définition de classe et <name> devient le nom d'une nouvelle classe ayant pour champs tous ceux de <superclass> accessibles par les fonction {<superclass>}:<fieldname> et pour nouveaux champs tous les <field>i accessibles par {<name>}:<fieldname>i. Nous dirons que <name> est une sous-classe de <superclass> et que <superclass> est la superclasse de <name>. On remarquera enfin que les instances de classes sont implémentées avec des *vectors* LE_LISP et non plus avec des cons.

```
? (setq client (cmakeq Client nom 'Jacquemart prenom 'Marcel pointure 48))
= #[Jacquemart Marcel () () 48]
? (setq suspect
?       (cmakeq Suspect nom 'Jacquemart
?                       prenom 'Marcel
?                       condamnations "Vol de chaussures"))
= #[Jacquemart Marcel () () Vol de chaussures]
? ({Individu}:nom client)
= Jacquemart
? ({Individu}:nom suspect)
= Jacquemart
? ({Client}:pointure client)
```

```
= 48
? ({Suspect}:condamnations suspect)
= Vol de chaussures
```

On notera de plus que si <class>1 est une sous-classe de <class> on a (omatchq <class> (omakeq <class>1 ...)) = t

Exemple:

```
? (omatchq Individu client)
= t
```

1.5 Propriétés Sémantiques des Records et des Classes

Nous sommes maintenant capables de définir des structures de données, nous aimerions définir des opérations, ou propriétés sémantiques, s'appliquant spécifiquement aux instances d'une telle structure. Reprenons l'exemple de la structure Point. Pour définir l'opération v**2 qui calcule la vitesse de ce point au carré, il suffit de faire sous CEYX:

```
? (de {Point}:v**2 (point)
      (+ (* ({Point}:vx point) ({Point}:vx point))
         (* ({Point}:vy point) ({Point}:vy point))))
= Point:v**2
? ({Point}:v**2 point2)
= 100
```

A ce point on peut considérer qu'on définit juste une fonction LISP ordinaire de nom {Point}:v**2. Nous allons un peu plus loin en définissant la construction semcall:

(semcall <structname> <sem> <arg>1 ... <arg>n)

Tous les arguments sont évalués. <structname> est un nom de classe ou de record, <sem> est un symbole. Si la propriété sémantique de nom <sem> a été définie pour <structname>, elle est appliquée aux arguments <arg>i. Sinon si <structname> n'a pas une définition de classe une erreur est déclenchée. Sinon si <structname> a une définition de classe et est extension d'une autre classe <superclass> on s'appelle récursivement avec <structname> = <superclass>. Sinon une erreur est déclenchée.

```
? (de {Individu}:print (individu)
?       (print "Nom: " ({Individu}:nom individu))
?       (print "Prenom: " ({Individu}:prenom individu))
?       t)
= #:Class:Individu:print
? ({Individu}:print client)
Nom: Jacquemart
Prenom: Marcel
= t
? (semcall 'Client 'print client)
Nom: Jacquemart
Prenom: Marcel
= t
```

Nous pouvons raffiner cette propriété sémantique sur la structure Client en définissant:

```
? (de {Client}:print (client)
?     ({Individu}:print client)
?     (print "Pointure: " ({Client}:pointure client))
?     t)
= #:Class:Individu:Client:print
```

```
? (semcall 'Client 'print client)
Nom: Jacquemart
Prenom: Marcel
Pointure: 47
= t
? (semcall 'Suspect 'print suspect)
Nom: Jacquemart
Prenom: Marcel
= t
```

Cette construction permet donc d'établir une hiérarchie sémantique calquant la hiérarchie structurelle introduite avec la structure classe. Si les objets étaient à même de reconnaitre de quelle structure ils sont instance, il n'y aurait même plus à lui passer l'argument donnant cette information et elle prendrait sa pleine puissance. Nous reviendrons sur ce point dans le prochain chapitre.

1.6 Un Exemple Développé

Nous donnons un exemple de programmes manipulant des positions et des régions rectangulaires dans un plan cartésien dont l'axe des x est vertical dirigé vers le bas et l'axe des y horizontal dirigé vers la droite (système de coordonnées de style écran).

Tout d'abord, nous définissons le record Position par:

```
(defrecord Position (x 0) (y 0))
```

Pour créer des instances de position, nous définissons:

```
(de Position (xpos ypos) (ormkeq Position x xpos y ypos))
```

Pour effectuer une translation sur une position, nous définissons la propriété sémantique translate:

```
(de {Position}:translate (pos dx dy)
    ({Position}:x pos (+ dx ({Position}:x pos)))
    ({Position}:y pos (+ dy ({Position}:y pos)))
    pos)
```

Ceci étant, nous introduisons la structure Region pour modéliser les régions rectangulaires. Une région est déterminée par son coin supérieur gauche et son coin inférieur droit qui sont des positions:

```
(defrecord Region upper-left bottom-right)
```

Et pour créer des instances de région à partir de deux positions:

```
(de Region (pos1 pos2)
    (let ((x1 ({Position}:x pos1))
          (y1 ({Position}:y pos1))
          (x2 ({Position}:x pos2))
          (y2 ({Position}:y pos2)))
         (ormkeq Region
                 upper-left (Position (min x1 x2) (min y1 y2))
                 bottom-right (Position (max x1 x2) (max y1 y2)))))
```

Nous définissons un certain nombre de propriétés sémantiques calculant des caractéristiques géométriques des régions:

```
(de {Region}:xmin (region)
    ({Position}:x ({Region}:upper-left region)))
```

```
(de {Region}:ymin (region)
    ({Position}:y ({Region}:upper-left region)))

(de {Region}:xmax (region)
    ({Position}:x ({Region}:bottom-right region)))

(de {Region}:ymax (region)
    ({Position}:y ({Region}:bottom-right region)))

(de {Region}:width (region)
    (1- (- ({Region}:xmax region) ({Region}:xmin region))))

(de {Region}:height (region)
    (1- (- ({Region}:ymax region) ({Region}:ymin region))))
```

Nous définissons maintenant une propriété sémantique des régions permettant de déterminer si une position donnée est à l'intérieur d'une région:

```
(de {Region}:contains-position (region pos)
    (not
      (or (< ({Position}:x pos) ({Region}:xmin region))
          (> ({Position}:x pos) ({Region}:xmax region))
          (< ({Position}:y pos) ({Region}:xmin region))
          (> ({Position}:y pos) ({Region}:ymin region)))))
```

Etant données deux régions, nous aimerions savoir si elles ont une intersection. Pour ce faire nous définissons la propriété sémantique intersects pour les régions, qui renvoie nil en valeur s'il n'y a pas intersection, la région d'intersection autrement.

```
(de {Region}:intersects (region1 region2)
    (let ((xmin1 ({Region}:xmin region1))
          (ymin1 ({Region}:ymin region1))
          (xmax1 ({Region}:xmax region1))
          (ymax1 ({Region}:ymax region1))
          (xmin2 ({Region}:xmin region2))
          (ymin2 ({Region}:ymin region2))
          (xmax2 ({Region}:xmax region2))
          (ymax2 ({Region}:ymax region2)))
        (setq xmin1 (max xmin1 xmin2)
              ymin1 (max ymin1 ymin2))
        (if (and (> (- (min xmax1 xmax2) xmin1) 0)
                 (> (- (min ymax1 ymax2) ymin1) 0))
            (Region (Position xmin1 ymin1)
                    (Position (min xmax1 xmax2) (min ymax1 ymax2)))
            nil)))
```

Enfin, nous pouvons définir des transformations sur les régions comme la translation:

```
(de {Region}:translate (region dx dy)
    ({Position}:translate ({Region}:upper-left region) dx dy)
    ({Position}:translate ({Region}:bottom-right region) dx dy)
    region)
```

et la transformation qui met le coin supérieur gauche d'une région sur une position donnée:

```
(de {Region}:move-to (region pos)
    ({Region}:translate region
            (- ({Position}:x pos) ({Region}:xmin region))
            (- ({Position}:y pos) ({Region}:ymin region))))
```

CHAPITRE 2

Les Records Autotypés

Ce chapitre suit un plan similaire à celui du chapitre précédent. Nous y définissons de nouvelles notions de records et de classes qui diffèrent des précédentes en ce que leurs instances portent toujours l'information du nom du record ou de la classe qui a permis de les engendrer. La seule construction réellement nouvelle de ce chapître est la construction send qui joue un rôle majeur dans CEYX.

2.1 Introduction

Reprenons l'exemple du chapître précédent sur les individus en définissant cette fois-ci la classe des individus par la nouvelle construction deftclass:

```
? (deftclass Individu nom prenom adresse telephone)
= #:Tclass:Individu
```

et créons maintenant une instance d'Individu par la construction omakeq:

```
? (setq individu (omakeq Individu nom 'Jacquemart prenom 'Marcel))
= #(#:Tclass:Individu . #[Jacquemart Marcel () ()])
```

Comme précédemment, nous avons accès aux divers champs de la structure Individu:

```
? ({Individu}:nom individu)
= Jacquemart
? ({Individu}:adresse individu)
= ()
```

La seule différence avec le chapître précédent est que maintenant les instances produites par omakeq portent l'information de la classe qui les a créées, information qui est accessible à l'utilisateur par la fonction model:

```
? (model individu)
= #:Tclass:Individu
```

Nous pouvons redéfinir les sous-classes Client et Suspect par:

```
? (deftclass {Individu}:Client pointure)
= #:Tclass:Individu:Client
? (deftclass {Individu}:Suspect condamnations)
= #:Tclass:Individu:Suspect
```

et créer des instances de ces classes:

```
? (setq client (omakeq Client nom 'Jacquemart prenom 'Marcel pointure 48))
= #(#:Tclass:Individu:Client . #[Jacquemart Marcel () () 48])
? (setq suspect
?       (omakeq Suspect
?               nom 'Jacquemart
?               prenom 'Marcel
?               condamnations "Vol de chaussures"))
```

```
= #(#:Tclass:Individu:Suspect . #[Jacquemart Marcel () () Vol de chaussures])
```

et vérifier que ces instances portent bien leur marque de fabrique:

```
? (model client)
= #:Tclass:Individu:Client
? (model suspect)
= #:Tclass:Individu:Suspect
```

et que client est bien un Individu:

```
? (omatchq Individu client)
= t
```

Comme précédemment, nous définissons une propriété sémantique d'impression pour les Individus:

```
? (de {Individu}:print (individu)
?     (print "Nom: " ({Individu}:nom individu))
?     (print "Prenom: " ({Individu}:prenom individu))
?     t)
= #:Tclass:Individu:print
```

Nous avons vu que cette propriété sémantique peut être déclenchée:

- soit par appel de la fonction LISP {Individu}:print sur un individu ({Individu}:print client),

- soit par invocation de semcall, (semcall 'Client 'print client), auquel cas la propriété sémantique sera recherchée d'abord dans celles de Client puis, en cas d'échec, dans les propriétés sémantiques des superclasses de Client, en l'occurrence Individu.

Compte tenu que les instances portent maintenant l'information du record ou de la classe qui a permis de les engendrer, nous pouvons, dans le cas des propriétés sémantiques qui prennent pour premier argument un objet qui est une instance du record ou de la classe associée, donner une nouvelle construction de déclenchement de propriétés sémantiques qui ne prend plus l'argument nom de record ou de classe. Cette construction appelée send prend en argument un nom de propriété sémantique <sem> et un nombre arbitraire d'autres arguments <arg>1 ... <arg>n et est équivalente à:

```
(semcall <sem> (model <arg>1) <arg>2 ... <arg>n)
```

Nous aurons ainsi:

```
? (send 'print client)
Nom: Jacquemart
Prenom: Marcel
= t
? (send 'print suspect)
Nom: Jacquemart
Prenom: Marcel
= t
```

On notera au passage que la propriété print des Individus est générique pour tous les Individus compte tenu de la dépendance hiérarchique des classes. Comme précédemment, nous pouvons raffiner la propriété print pour les Clients:

```
? (de {Client}:print (client)
?     ({Individu}:print client)
?     (print "Pointure: " ({Client}:pointure client))
?     t)
= #:Tclass:Individu:Client:print
```

et cette fois-ci nous aurons:

```
? (send 'print client)
Nom: Jacquemart
Prenom: Marcel
Pointure: 48
= t
```

Définissons maintenant une propriété sémantique d'impression pour les Suspects:

```
? (de {Suspect}:print (suspect)
?     (print "Le denomme " ({Individu}:nom suspect)
?            " " ({Individu}:prenom suspect)
?            " a ete condamne pour " ({Suspect}:condamnations suspect))
?     t)
= #:Tclass:Individu:Suspect:print
? (send 'print suspect)
Le denomme Jacquemart Marcel a ete condamne pour Vol de chaussures
= t
```

L'un des intérêts majeurs de la construction send est donc qu'elle se substitue élégamment à des constructions de type selectq quand on veut effectuer des actions différentes suivant le type de l'objet qu'on manipule.

2.2 Définition d'un Record Autotypé ou Trecord

Un record autotypé est défini à l'aide de la forme deftrecord:

(deftrecord <name> <field>1 ... <field>n)

<name> est un symbole qui devient le nom de la structure ainsi définie et les <field>i sont de la forme <symbol>i ou (<symbol>i <init>i), où <symbol>i est le nom du champ et <init>i, qui est évaluée est la valeur d'initialisation du champ. Les champs pour lesquels la valeur d'initialisation n'est pas donnée sont initialisés à nil.

Cette construction associe au symbole <name> la définition de Trecord à n champs de noms <symbol>i. De plus n fonctions de nom {<symbol>}:<symbol>i sont engendrées pour permettre d'accéder aux instances de cette structure en lecture et en écriture.

La seule différence entre Record et Trecord est que les instances des seconds (obtenues par appel de (omakeq <name> ...)) portent l'information du Trecord qui a servi a les engendrer. Les fonctions omakeq, omatchq, oputq et ogetq fonctionnent sur ces nouveaux objets,

2.3 Définition d'une Classe Autotypée ou Tclasse

Une classe autotypée est définie à l'aide de la forme deftclass:

(deftclass <name>/{<superclass>}:<name> <field>1 ... <field>n)

Dans le premier cas, cette forme est équivalente au deftrecord, sinon qu'on définit cette fois-ci un Trecord extensible. Dans le second cas <superclass> doit avoir une définition de Tclasse et <name> devient le nom d'une nouvelle classe ayant pour champs tous ceux de <superclass> accessibles par les fonction {<superclass>}:<fieldname> et pour nouveaux champs tous les <field>i accessibles par {<name>}:<fieldname>i. Nous dirons que <name> est une sous-classe de <superclass> et que <superclass> est la superclasse de <name>.

De même que pour les Trecords la seule différence entre Classe et Tclasse est que les instances des seconds (obtenues par appel de (omakeq <name> ...)) portent l'information de la Tclasse qui a servi a les engendrer.

2.4 La Construction send

Les propriétés sémantiques des Trecords et des Tclasses sont définies de la même manière que pour les Records et les Classes. La construction semcall fonctionne également sur les Trecords et les Tclasses.

(send <sem> <object> <arg>1 ... <arg>n)

Tous les arguments sont évalués, <object> doit être une instance de Trecord ou de Tclasse. Cette construction ne fonctionne que pour les propriétés sémantiques dont le premier argument est une instance de la structure pour laquelle on définit cette propriété sémantique. Elle est équivalente à:

```
(semcall <sem> (model <object>) <object> <arg>1 ... <arg>n)
```

On trouvera de nombreux exemples d'application dans le manuel programmeur CEYX: "Programmer en CEYX".

Index des fonctions LISP

Table des matières

Imprimé en France

par

l'Institut National de Recherche en Informatique et en Automatique